8.8 Testing Functions IV: Efficiency

We hope that the number of sections of these notes dedicated to testing demonstrates its importance in the process of software development. What is perhaps surprising is that testing is not limited to correctness. In fact, strict efficiency constraints are the norm in several domains. For example, a Playstation controller must send wireless signals at the touch of a button or the move of a joystick—if the function for doing so were correct, but took 10 seconds, players would not be happy. Similarly, a search on Google that sifts through terabytes of data must also be fast.[1] In this section, we will discuss how to write tests for efficiency of functions.

[1] Check it out: each search you do on Google reports how many results were found in how many fractions of a second.

An efficiency test

Earlier we saw how to use the `timeit` module to measure the time taken to execute a piece of Python code. Let's see how we might setup a performance constraint using `timeit` and our implementation of `is_prime`:

```
from math import floor, sqrt
from timeit import timeit


def is_prime(p: int) -> bool:
    """Return whether p is prime."""
    possible_divisors = range(2, floor(sqrt(p)) + 1)
    return (
        p > 1 and
        all(not p % d == 0 for d in possible_divisors)
    )


def test_is_prime_performance() -> None:
    """Test the efficiency of is_prime."""
    numbers_to_test = range(2, 1000)
    for number in numbers_to_test:
        time = timeit(f'is_prime({number})', number=100, globals=globals())
        assert time < 0.001, 'Failed performance constraint of 0.001s.'
```

There are several issues here that we need to keep in mind. The performance constraint of 0.001 seconds is for the total runtime of 100 calls to `is_prime` for only one number in `numbers_to_test` (there will be as many assertions as there are elements in `numbers_to_test`).

Where did the argument `number=100` come from? Should it be more or less? An important thing to remember is a computer system is not at all like a science experiment you would setup in a chemistry or biology lab. There are too many external factors (i.e., background processes being run) that can impact the results. To avoid this issue, several samples of an experiment (i.e., measurements of time) need to be taken. The field of statistics can help inform us on whether or not 100 samples is sufficient for this test.

Next, where did `0.001` seconds come from? The number is most certainly arbitrary in this example. Computer systems are very different from one another, in terms of both hardware and software. While the assertions may hold for all `numbers_to_test` on one computer, they may not hold on another. The `0.001` seconds may be tuned over time in the testing suite. Or it can help identify the minimum hardware requirements for running a piece of software.

While it is easy to write the Python code that checks for performance, coming up with the actual parameters (number of function calls, inputs to the function, total acceptable runtime) is quite challenging, and often domain-dependent. For example, in user interfaces, a great deal of research has gone into how fast actions should be; a so-called "instantaneous" action in a user interface should complete in 0.1 seconds. Other domains, such as embedded systems, have a series of functions that must meet hard deadlines in order for the computer system to function properly (e.g., in a spaceship).

But what about domains where there are no guidelines or standards? Runtime constraints that are tuned over time can still be useful in discovering changes in program efficiency due to bug fixes or new features. When a code change causes an efficiency test to fail, the programmers can decide whether to the efficiency constraint or explore alternative code changes. Without efficiency tests in place, the change in performance might not have been found until it impacted a real user of the software!